Future Generation Computer Systems 160 (2024) 420–432

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# Scalable I/O aggregation for asynchronous multi-level checkpointing

Mikaila J. Gossman [a,*], Bogdan Nicolae [b], Jon C. Calhoun [a]

[a] Holcombe Department of Electrical and Computer Engineering, Clemson University, Clemson, 29631, SC, USA
[b] Mathematical and Computer Science Division, Argonne National Laboratory, Lemont, 22222, IL, USA

ARTICLE INFO

*Keywords:*
Checkpoint-restart
Asynchronous I/O
Distributed I/O aggregation

ABSTRACT

Checkpointing distributed HPC applications is a common I/O pattern with many use cases: resilience, job management, reproducibility, revisiting previous intermediate results, etc. This is a difficult pattern for a large number of processes that need to capture massive data sizes and write them persistently to shared storage (e.g., parallel file system), which is subject to I/O bottlenecks due to limited I/O bandwidth under concurrency. In addition to I/O performance and scalability considerations, there are often limits that users impose on the number of files or objects that can be used to capture the checkpoints. For example, users need to move checkpoints between HPC systems or parallel file systems, which is inefficient for a large number of files, or need to use the checkpoints in workflows that expect related objects to be grouped together. As a consequence, *I/O aggregation* is often used to reduce the number of files and objects persistent to shared storage such that it is much lower than the number of processes. However, I/O aggregation is challenging for two reasons: (1) if more than one process is writing checkpointing data to the same file, this causes additional I/O contention that amplifies the I/O bottlenecks; (2) scalable state-of-art checkpointing techniques are asynchronous and rely on multi-level techniques to capture the data structures to local storage or memory, then flush it from there to shared storage in the background, which competes for resources (I/O, memory, network bandwidth) with the application that is running in the foreground. State of art approaches have addressed the problem of I/O aggregation for synchronous checkpointing but are insufficient for asynchronous checkpointing. To fill this gap, we contribute with a novel I/O aggregation strategy that operates efficiently in the background to complement asynchronous C/R. Specifically, we explore how to (1) develop a network of efficient, thread-safe I/O proxies that persist data via limited-sized write buffers, (2) prioritize remote (from non-proxy processes) and local data on I/O proxies to minimize write overhead, and (3) load-balance flushing on I/O proxies. We analyze trade-offs of developing such strategies and discuss the performance impact on large-scale micro-benchmarks, as well as a real HPC application (HACC).

## 1. Introduction

Checkpointing is a fundamental pattern used by a variety of applications at both small and large scales. Widely adopted for resilience purposes (e.g., restart the application from a past reliable consistent state to minimize the amount of lost computation in case of failures), it has seen an explosion of use cases that enable applications to progress faster and achieve shorter time-to-solution even in the absence of failures. For example, adjoint computations (essential in financial modeling, weather prediction, computational fluid dynamics, seismic imaging, control theory) [1] need to capture a history of checkpoints in a forward pass, which are then revisited in a backward pass. AI training, increasingly used by scientific applications, often takes trajectories that do not lead to convergence or learn undesirable patterns, prompting the need to backtrack to an earlier checkpoint of the learning model to try an alternative (reconfigure the AI model or change the training trajectory). This effect is especially exacerbated in large models, such as large language models (LLMs) and transformers [2]. Transfer learning and fine-tuning [3] use checkpoints of AI models trained in a different context to construct new AI models with a better starting point than training from scratch, which accelerates the training and/or achieves better quality metrics. Checkpoints are also instrumental in enabling malleability for tightly coupled applications (e.g., HPC simulations such as LAMMPS molecular dynamics [4] or data-parallel AI training [5,6]): suspend the execution of distributed processes by taking a checkpoint, then restart in a different configuration (different number of processes, different data distribution, etc.). This enables the application to elastically allocate the resources needed to finish faster and/or reduce resource utilization. Many other use cases are reported by the scientific

* Correspondence to: Holcombe Department of Electrical and Computer Engineering, Clemson University, Clemson, SC, USA.
*E-mail addresses:* mikailg@g.clemson.edu (M.J. Gossman), bnicolae@acm.org (B. Nicolae), jonccal@clemson.edu (J.C. Calhoun).

https://doi.org/10.1016/j.future.2024.06.003
Received 2 January 2024; Received in revised form 21 May 2024; Accepted 3 June 2024
Available online 7 June 2024

community: suspend-resume (e.g., to preempt a batch job in favor of a higher priority job), migration (checkpoint on one machine, restart on another), debugging (replay a problematic code region to avoid running from scratch), reproducibility (checkpoint and compare intermediate data during repeated runs).

Checkpointing usually involves a distributed set of application processes that concurrently capture representative data structures as checkpoints that are persisted to external storage (e.g., a parallel file system [7] or PFS), from where they are reused later. On a typical HPC machine, the aggregated I/O bandwidth to the external storage is limited. For this reason, synchronous checkpointing strategies that block applications until all processes checkpoint to a PFS are impractical at large scale due to I/O bottlenecks [8–10]. To alleviate this issue, asynchronous multi-level checkpoint systems (e.g., VELOC [11]) and I/O libraries (e.g., ADIOS2 [12]) write the checkpoints to fast, node-local storage (e.g., SSDs), then flush them from there to the PFS in the background, while the application continues running, thereby masking the I/O bottlenecks.

As such, asynchronous checkpointing strategies share the resources of the compute nodes (CPU cores, memory, network bandwidth) with the application processes, which creates contention and interferes with the application progression and performance. Thus, on one hand, it is important to minimize this interference by prioritizing the application over the background flushes. On the other hand, the increasing checkpoint frequency prompts the need to reduce the duration of the flushes, to avoid a situation where checkpoints are produced faster on the local storage than they can be flushed to the PFS. To optimize this trade-off, checkpointing strategies adopt a one-file-per-process asynchronous flush strategy [11,13,14], which has several advantages: (1) it is simple, portable and efficient (no I/O locking needed, which otherwise would be needed to maintain file consistency under parallelism); (2) it enables an independent strategy for mitigating contention on each compute node; (3) it takes advantage of PFS fine-tuning typically adopted on HPC machines that favor load balancing across the processes; (4) it avoids *false sharing*, which can occur when I/O belonging to different processes unnecessarily compete for the same PFS allocation unit (e.g., stripe of a shared file).

However, one-file-per-process flush strategies are not without limitations. At large scale, the number of files can explode, which in some cases overwhelms the PFS metadata servers. Specifically, a PFS experiences metadata bottlenecks when large numbers of files are accessed simultaneously (especially when stored in the same directory) [7]. Furthermore, it is difficult for developers to manage a large number of checkpoint files, especially in scenarios that involve: moving between data centers, verification of integrity, use in a producer–consumer workflow, etc. To alleviate these limitations, I/O aggregation (e.g., as implemented by GenericIO [15]) can be used to combine the checkpoints from $N$ processes into $K < N$ files, such that the number of checkpoint files becomes manageable without compromising the ability to efficiently leverage parallel I/O operations from multiple compute nodes.

Although effective for synchronous checkpointing [15–17], such I/O aggregation strategies remain largely unexplored for asynchronous checkpointing. In this paper, we aim to fill this gap. Unfortunately, adapting I/O aggregation strategies for asynchronous checkpointing is challenging for several reasons: (1) the use of tightly coupled communication patterns that synchronize and block all processes (in the spirit of MPI-IO collectives) is not feasible; (2) I/O aggregation is a background activity, therefore the competition with the application for resources (interference) needs to be minimized; (3) the advantages of one-file-per-process asynchronous strategies (simplicity, portability, load balancing, no locking or false sharing) are not implicitly inherited and need to be achieved through alternative techniques; (4) there is a need for flexibility and configurability to efficiently meet the needs of diverse HPC workloads on diverse HPC machines.

Our contribution proposes the design and implementation of an I/O aggregation technique specifically designed for asynchronous multi-level checkpointing that addresses the aforementioned challenges. Our key idea is to simultaneously co-optimize multiple trade-offs: maximize I/O parallelism to the PFS for distributed flushes of checkpointing data without overwhelming the PFS at either the metadata level or by excessive competition for limited aggregated I/O bandwidth, while at the same time enabling a decoupled design suitable for low-overhead background I/O activity that facilitates asynchronous operation with low application interference. We summarize our contributions as follows:

1. We design an optimized multi-threaded aggregation strategy to complement asynchronous multi-level checkpointing. To this end, we proposed a configurable trade-off between issuing concurrent I/O and managing over-subscription of both PFS (e.g., I/O servers, storage devices, available I/O bandwidth) and node-local (memory, CPU cores, bandwidth) resources (Section 4.1)
2. We introduce a producer–consumer workflow via an in-memory queue using dedicated I/O threads which: (1) overlap various I/O operations, thereby maximizing parallelism, and (2) employ a low-overhead synchronization scheme thanks to a decoupled design (Section 4.3).
3. We implement a flexible solution that enables application developers to fine-tune numerous parameters (e.g., number of files, number of I/O threads, maximum buffer size, data distribution among I/O threads, etc.) to adapt to a variety of HPC workloads and systems (Section 4.4).
4. We evaluate our proposal in comparison with two state-of-the-art I/O libraries that implement distributed I/O aggregation strategies. To this end, we designed and developed a custom checkpointing benchmark to perform targeted evaluations that complement evaluations with real-world scientific applications. Results at scale show our proposal achieves a minimum of 1.6× higher flush throughput than the two state-of-the-art aggregation strategies using synthetic checkpointing benchmark, and 22% higher flush throughput with the real-world scientific application (Section 5).

## 2. Motivation

File-per-process checkpointing techniques, when applied at Exascale, results in a large number of checkpoint files written concurrently to the PFS. However, despite decades of evolution, parallel file systems commonly used by HPC systems (e.g., Lustere [7]) have seen relatively little evolution. Subject to the aging POSIX standard, a PFS is subject to metadata bottlenecks, especially when a large number of files is written to the same directory. The advent of AI and ML applications that commonly complement traditional HPC applications has further exacerbated this bottleneck. For example, large language models (LLMs) need to be checkpointed frequently during lengthy pre-training at large scale (tens of thousands of GPUs) not only to survive failures or for administrative purposes (e.g., to suspend the training and resume it later in order to run other jobs), but also due to loss spikes, biases and undesirable learning patterns (which are mitigated by rolling back to past checkpoints, adjusting the model or training data, and resuming the training). In this case, multiple data structures need to be checkpointed on each GPU (model parameters, optimizer state), each of which is typically written as a separate file, resulting in an explosion of the number of files at scale [18]. In addition to metadata bottlenecks, some HPC systems only allow applications to have a certain number of files open at a time or otherwise risk severely degrading the system [19]. For sufficiently large workloads, checkpointing and restarting from file-per-process formats are therefore limited by this cap, which may be significantly lower than the number of processes employed by the application. Thus, file-per-process flushing strategies

are incompatible with the current trajectory of scientific computing and workloads.

Beyond performance, I/O aggregation is necessary from a user manageability standpoint. In this context, file-per-process checkpointing formats make it very difficult for users to easily list, analyze, or migrate said checkpoints [15]. Thus, aggregating checkpoints into a more compact layout makes it simpler for users to perform the aforementioned tasks. Furthermore, aggregating groups of files is also shown to provide higher compression ratios [16], greatly reducing the storage overhead, which is a necessary task of big data workloads. Thus, efficient methods of file aggregation are increasingly sought-after by users, even if metadata performance aspects are not a concern.

*Problem formulation.* We assume $N$ processes, distributed across $M$ compute nodes. The processes need to simultaneously capture critical data structures into a globally consistent checkpoint. The sizes of the data structures captured by each process may vary. The application uses an asynchronous multi-level checkpointing runtime that blocks only until each process has written its data structures as a file on the local storage (host memory or NVMe) of the compute node that is hosting the process. Afterwards, the application is allowed to resume. Meanwhile, in the background, the checkpointing runtime flushes the $N$ local files to the PFS. We aim to design an asynchronous I/O aggregation strategy that simultaneously satisfies the following objectives: (1) it produces at most $K$ files on the PFS, with $K$ configurable and specified by the user; (2) it minimizes the end-to-end impact on the application, which is measured as the increase in runtime due to checkpointing (compared with no checkpointing).

## 3. Background and related work

*Simple I/O aggregation.* A straightforward way to perform I/O aggregation is to concatenate the checkpoints of multiple processes into larger files. In this case, multiple concurrent writers are allocated disjoint regions of shared files (each starting at a fixed offset) and simply write into these regions without the need for synchronization. In this case, the competition between the writers needs to be handled efficiently by the parallel file system (PFS) itself. It is often the case that the regions are misaligned with respect to the preferred layout of the PFS for large shared files (e.g., striping into fixed-sized chunks), which leads to *false sharing* (two regions sharing the same chunk and causing write competition) and/or excessive distribution (regions split across multiple chunks resulting in the same writer having to interact with multiple I/O servers). In some cases, this contention can be mitigated by configuring the data layout in the file system. However, this requires complex organization and exchanging of data among processes. Paired with the fact that parallel I/O requests are not always processed in an optimal order, performance rapidly degrades not only for the application but for the file system as a whole [20].

Thus, more complex strategies that mitigate competition for PFS resources are needed. The most common is collective *two-phase I/O* [21], originally introduced in the MPI-2 standard. In this case, a first phase is responsible for collecting I/O requests on designated proxy processes (denoted aggregators), which is then followed by a second phase in which the proxies consolidate and run the I/O requests on behalf of the other MPI ranks using an optimized configuration [22] that is preferred by the PFS and that achieves higher I/O performance and scalability compared with the simple I/O aggregation strategy [23–25].

*MPI-IO.* Variations of the two-phase I/O have been explored and incrementally improved for MPI-I/O implementations like ROMIO [26]. They are exposed as both blocking (synchronous) and non-blocking (asynchronous) I/O collectives to the applications. In the context of checkpointing, one could imagine simply calling synchronous MPI-IO collectives from background I/O threads on different communicators to provide asynchronous checkpointing capabilities. However, such a solution is subject to limitations as it requires the MPI implementation to have optimized multi-threaded support and it does not enable any form of control over the contention between the application and the background I/O threads. On the other hand, large non-blocking MPI-IO collectives often lead to excessive resource utilization (e.g., buffering of intermediate data on the aggregators), which is why they are typically broken into a chain of smaller I/O requests. However, since each invocation is a collective, this leads to excessive synchronization overheads and therefore poor I/O performance and scalability [27,28].

To address these limitations, MPI-IO implementations [27] have explored the idea of overlapping the two phases, which better utilizes the available resources and increases the I/O throughput. This approach was expanded [28] with a second layer of threading to pipeline reading, exchanging, or writing data. Such approaches may mask synchronization overheads but still suffer from excessive resource utilization. Furthermore, they are not flexible enough to adapt to the preferred I/O parallelism of the PFS, often leading to over-subscription through excessive threading or under-utilization of the I/O resources through insufficient threading.

Locality-aware optimizations to improve the data exchange (phase 1) of two-phase I/O are also possible. For example, node-local data aggregation [29] reduces both the number of messages exchanged between the aggregators and the rest of the MPI ranks, as well as the amount of concurrent communication (since not all ranks participate in the collective). TAPIOCA [22] forms topology-aware groups, electing a leading compute node in each group that uses a pipeline approach with two buffers to overlap the collection of data from the rest of the MPI ranks in the group with the flushing of the data to the PFS. Their results show more than 2× higher I/O throughput compared with baseline MPI-I/O collectives. Such topology-aware optimizations are complementary to multi-level asynchronous checkpointing and can be leveraged for additional I/O performance and scalability optimization.

*I/O request consolidation.* Solutions like FILCIO [30] intercept I/O calls from HPC applications at runtime. These intercepted I/O requests get buffered in memory up to a predefined size. When the buffers are filled up, fewer consolidated (and more efficient) I/O requests are issued to the PFS. Such solutions complement I/O aggregation from a different perspective: instead of reducing the number of output files, they reduce the number of interactions with the PFS, which reduces latency and increases the I/O throughput without affecting the number of files.

*I/O runtimes with support for aggregation.* ADIOS2 [12] introduced their own blocking and non-blocking I/O primitives specifically designed for HPC applications. These I/O primitives can be leveraged to provide asynchronous checkpointing capability. Specifically, the non-blocking mode of operation buffers the data on local storage, allows the application to continue, then flushes the data from local storage to the PFS using a variant of two-phase I/O. Paired with a number of malleable parameters (e.g., maximum amount of I/O threads, maximum buffer size, number of aggregators, etc.), ADIOS2 provides a flexible, multi-threaded I/O solution that can be adapted to a variety of HPC workloads and systems. However, its focus on a detached data layout and organization (which is separately configurable) introduces additional complexity in the context of checkpointing, which may lower I/O performance and scalability.

GenericIO [15] is a popular I/O library, especially in the cosmology community. It enables configurable data aggregation through a pluginable architecture that supports both MPI-IO (two-phase aggregation strategy) and direct POSIX I/O (simple aggregation strategy). It offers flexible customization options and also automates the selection of optimal parameters, including aggregation strategy and number of output files. However, it only offers support for synchronous I/O.

HDF5 [31] offers support for both asynchronous I/O and data aggregation on top of low-level POSIX API, which can be used to write structured, self-descriptive data to a PFS. In this regard, HDF5 introduces a low-level storage abstraction that understands a basic read/write API, which can be used to implement different multi-level

I/O strategies, including I/O aggregation. Our approach can complement I/O libraries like HDF5 by offering specialized support for asynchronous aggregation of checkpoints.

*HPC checkpointing systems.* Several efforts leverage multiple storage levels in the context of checkpointing. For example, Scalable Checkpoint/Restart (SCR) [14], introduces multi-level resilience strategies that take advantage of the multiple storage levels: it supports local storage, partner replication and XOR encoding on remote nodes in addition to flushing the checkpoint data to the PFS. Fault Tolerant Interface (FTI) [32] is another related effort that offers similar support while adding Reed–Solomon (RS) encoding [33]. Both offer limited support for asynchronous checkpoint flushes between the levels and they adopt a one-file-per-process approach, which is subject to the limitations discussed in Section 1.

VELOC [34] is a checkpointing system that improves on the multi-level resilience strategies introduced by SCR and FTI. Specifically, it focuses on providing efficient asynchronous support to mask the overhead of the multi-level resilience strategies through a background engine that implements a modular checkpointing pipeline designed to streamline the entire life-cycle of the checkpoints, from serialization of checkpointing data, to additional transformations, integrity verification and checksumming, caching and aggregation. It mitigates interference with the application through a series of multi-threading policies that prioritize application resource utilization under competition. However, the default checkpoint aggregation strategy of VELOC only supports a single output file and is based on the simple offset-based strategy that leverages the POSIX API.

## 4. Proposed approach

In this section, we discuss our contribution: an I/O aggregation strategy specifically designed for multi-level asynchronous checkpointing. To this end, we design and implement a streamlined two-phase I/O strategy that overcomes the limitations of the approaches discussed in Section 3.

### 4.1. Design principles

*Leader election to optimize PFS I/O contention.* HPC systems feature a high compute node to I/O node ratio. For example, on Exascale ready machines like Frontier [35], the ratio of compute nodes to PFS I/O servers is 10:1. Thus, the overall I/O bandwidth of the PFS is under heavy contention, especially for I/O patterns that require all MPI ranks to simultaneously write data to the PFS (which is the case of checkpointing). Too much contention for I/O bandwidth reduces the performance and scalability of the PFS [36]. On the other hand, if not enough I/O parallelism is used, then the I/O bandwidth of the PFS remains under-utilized. Thus, there is a sweet spot in leveraging just enough I/O parallelism to fully utilize the PFS I/O bandwidth without causing excessive contention.

To address this challenge, we propose the following principle. First, we group the MPI ranks together into $K$ groups, with $K$ a configurable parameter. Each group is allowed to flush a single aggregated output file to the PFS that aggregates the checkpoint data from all MPI ranks of the group (resulting in a total of $K$ output files). Then, each group is independently responsible to elect a leader that will collect all checkpointing data from all MPI ranks of the group and will perform the flushes to the PFS on behalf of the group. Similar to [21,29], the leader represents both remote MPI ranks (residing on a different compute nodes) and local MPI ranks (residing on the same compute node). Only the remote MPI ranks need to send the checkpointing data to the leader, since the leader can implicitly access the checkpointing data of the local MPI ranks through the shared local storage. Using this approach has two advantages: (1) each group can act autonomously and can self-organize based on the checkpointing data distribution in order to optimize the communication between the leader and the rest of the remote MPI ranks; (2) since the I/O activity is centralized in a single process, it can be closely coordinated and fine-tuned through multi-threading using low-overhead synchronization (as opposed to inter-process or even inter-node synchronization, both of which have higher overhead).

*Streamlined producer–consumer flushing.* After each MPI rank has captured the checkpointing data to the local storage of its hosting compute node, the application continues running, while each I/O leader flushes the checkpointing data of its group asynchronously in the background to the PFS. State-of-art two-phase I/O aggregation strategies simply wait for the remote MPI ranks to send their checkpointing data to the local storage of the compute node hosting the leader, then, once all checkpointing data is available on the local storage, start writing to the PFS. Such a strategy is easy to implement and provides a clear separation of the two phases (which makes it easier to recover from failures), but, on the other hand, it has two important limitations: (1) it needs sufficient free space available on the local storage of the compute node hosting the leader in order to collect the checkpointing data from all MPI ranks of the group; (2) it does not overlap the collection of the checkpointing data from the remote MPI ranks with the flushes to the PFS, which limits the I/O performance and scalability.

To address this issue, we propose a streamlined producer–consumer solution based on circular buffer reuse: a limited number of fixed-sized buffers are reserved on the compute node hosting the leader to receive the checkpointing data from the remote MPI ranks (producers). The buffers are filled in a first-come first-served fashion. The size of each buffer and the number of buffers can be fine-tuned in conjunction with the group size to avoid starvation of the producers. At the same time, the I/O leader (consumer) constantly flushes the checkpointing data to the aggregated file on PFS. It prioritizes the checkpointing data available in the buffers filled by the remote MPI ranks, which releases the buffers back to the consumers at the earliest possible moment, thereby avoiding stalls. Only when no checkpointing data is available in the buffers, then it proceeds to flush checkpointing data from the local MPI ranks sharing the same compute node. Thanks, to this streamlined approach, we solve the two limitations of two-phase I/O simultaneously: (1) the space utilization of local storage reserved for receiving the remote checkpointing data is limited to a configurable fixed size; (2) the collection of checkpointing data and the flushes to the PFS fully overlap with minimal stalls (thanks to prioritization of remote checkpointing data).

*Optimized multi-threaded writes to the PFS.* It is important to note that many PFS deployments commonly used on leadership class HPC systems (e.g., based on Lustre [7]) cannot saturate the I/O bandwidth available between a compute node and an I/O server with a single large I/O request (e.g., a write of GBs of data). Instead, several concurrent I/O requests are needed to saturate the I/O bandwidth. However, just like at global level, there is a competition trade-off in this case as well: spawning too many I/O threads to issue concurrent I/O requests may create excessive competition that saturates the I/O bandwidth but uses it sub-optimally, resulting in longer duration of I/O requests for each thread. Furthermore, in our case, each I/O leader needs to saturate the I/O bandwidth with concurrent write requests to the *same shared file*. In this case, additional considerations are important, such as alignment of write requests to offsets that are multiples of the allocation units used by the PFS, as well as maintaining locality of writes (to avoid concurrent interleavings that constantly jump to different offsets far apart from each other). Ignoring such considerations results in significant degradation of the I/O performance and scalability.

To address these issues, we build on our previous work that studies the problem of how to build a multi-threaded I/O solution that saturates the I/O bandwidth between a compute node and an I/O server of the PFS [37]. Specifically, the I/O leader spawns a controllable number of background I/O threads responsible for collaborating to

asynchronously flush the checkpointing data to the same shared file on the PFS. The optimal number of I/O threads can be determined experimentally using micro-benchmarks. This needs to be done only once at small scale and will ensure saturation of the I/O bandwidth between the compute nodes hosting the leaders and the I/O servers hosting the aggregated checkpoint files. Then, since the I/O leader prioritizes the received checkpointing data from remote MPI ranks (and this data can arrive in any order, potentially breaking locality), we adopt an append-only concurrent write strategy that allocates increasing offsets to the write requests of the I/O threads, while recording the distribution later in separate metadata. Such an approach is similar in spirit to log-structured write strategies, but without mixing the metadata and data streams and instead consolidating the metadata separately. This has an additional advantage in that the write requests can be easily aligned to offsets that are multiple of allocation units (unlike the case when metadata and data are mixed together). Finally, it is important to remember that concurrent write requests are overlapped with receiving the checkpointing data from the remote MPI ranks, both of which share the same network interface. However, modern network interfaces have separate physical send and receive links, which justifies the saturation of the I/O bandwidth with write requests. Furthermore, the I/O saturation is only beneficial up to a certain scale, after which the aggregated I/O bandwidth of the whole HPC machine becomes a bottleneck. Our approach supports a configurable number of I/O threads per leader that can be fine-tuned based on both saturation and scale considerations.

*Load balancing among I/O leaders.* The I/O leaders may be subject to load imbalance, both across groups and within groups. This can happen for several reasons: (1) the MPI ranks may need to checkpoint data structures of different sizes; (2) the MPI ranks checkpoint data structures of equal sizes but apply compression to reduce the checkpoint sizes, which results in different compression ratios; (3) the MPI ranks apply post-processing transformations (e.g., filters) that reduce the size of the checkpointing data. Since a checkpoint is considered to be successfully written to the PFS only after all I/O leaders have finished, load imbalance may lead to stragglers, which negatively impacts the overall asynchronous flush throughput.

To address this issue, we leverage two observations. First, if the groups are large enough, there is a high statistical probability that the total checkpointing data of the groups are relatively close to each other, which happens because the developers tend to partition their problem into subdomains of equal complexity. Thus, we do not aim to perform load balancing across groups and rather prefer to implement a simple scheme that preserves the autonomy of the groups and avoids synchronization among them. However, it is important to note that should the need arise for load balancing across groups, our proposal can be easily extended to support it (e.g., groups of different number of MPI ranks can be formed using low-overhead synchronization such as prefix-sum). However, within each group, we employ load balancing using a secondary leader election protocol that chooses the compute node with the largest amount of local checkpointing data to host the I/O leader. Using this approach, the amount of checkpointing data that needs to be sent by the remote MPI ranks is minimized, thereby reducing potential stalls due to producers not filling the buffers fast enough for the I/O threads of the leader.

### 4.2. Leader election

In this section, we zoom on the leader election algorithm designed to control the communication between the large set of application processes and the PFS. As such, we combine optimization features from previous works such as two-layer aggregation [29] and informed elections based on the data layout [22].

Specifically, each compute node uses a single MPI process (representing all other MPI processes co-located on the same compute node) to run Algorithm 1, we call these the *representative ranks*. Algorithm 1 partitions the representative ranks into $K$ groups of roughly equal size (line 3), where representative ranks in proximity to each other are assigned to the same group, which has the best chance of accommodating the dragonfly topology. After this step, the groups are considered autonomous and only need to perform inter-group communication during the election policy.

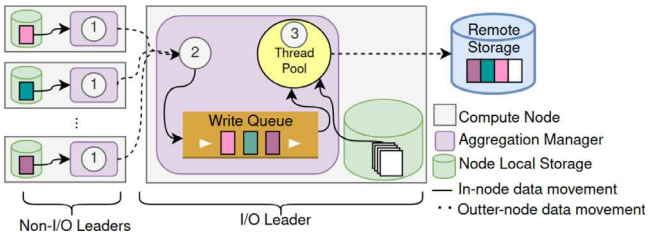


**Fig. 1.** I/O leaders processing received and local checkpointing data.

**Algorithm 1:** I/O Leader Election

**Input:** $K$: # of I/O leaders (and groups); $C$: total # of compute nodes employed by application; $C_i$: a normalized ID of a compute node; $S$: aggregate size of checkpoint data residing on compute node $C_i$; $R$: Role structure $\langle isLeader = false, remote = 0, local = S \rangle$

```
1  Function leader_election(K, C, C_i, S, I):
2      G_ID ← Group ID
3      G ← communicator group { C | (⌊C_i/⌈C/K⌉⌋ = G_ID) }
4      P ← ⟨C_i, S⟩ pair
5      A ← MPI_Allgather( P ∈ G)
6      L ← policy(A)
7      if L == C_i then
8          R.isLeader ← true
9          R.remote ← sum(A.S) − S
10 Function policy(A):
11     max_pairs ← {P ∈ A | P.S = max{A.S}}
12     L ← min(max_pairs.C_i)
13     return L
```

In our election policy, representative ranks in the same group collectively exchange ID's and the aggregate size of data they contribute to the shared checkpoint file, favoring the compute node whose local checkpoint data is the largest (to minimize the amount of checkpointing data exchanged between I/O leaders). In the case when there is a tie (e.g., more than one max element in $A$), we choose the lowest MPI rank (e.g., first element in $A$ where $P.S = \texttt{max}$). Simultaneously, the compute node elected as the leader then knows the amount of data to receive from all other remote MPI ranks, and then establishes a stream-lined producer–consumer workflow, which we discuss next.

### 4.3. Streamlined multi-threaded producer–consumer flushing

This section details our implementation of the streamlined multi-threaded producer–consumer flush strategy. Fig. 1 provides a high-level illustration of our implementation. The I/O threads spawned on each compute node (by the representative MPI rank described in the leader election), get assigned as either a: (1) sender, (2) receiver, or (3) writer (numbered white circles in Fig. 1). We have producer MPI ranks (e.g., non-leaders) employ a single sender, and consumer MPI ranks (e.g., I/O leaders) a single receiver. While we could employ more senders and receivers, the size constraints implemented by leaders to moderate resource consumption make it more advantageous to allocate more resources to writers, which will be the most impeding factor to producer ranks completing sends.

The algorithms for each type of background I/O thread are listed below. For clarity, Table 1 lists variables that are either thread-local

**Table 1**
Table of variables used in I/O threads.

| Variable | Description |
|---|---|
| $BS$ | block size of buffer (assigned by user) |
| $B$ | buffer of size $BS$ |
| $S$ | **shared** aggregate size of local data |
| $\langle src, l\_o \rangle$ | **shared** offset in local (on-node) data |
| $r\_o$ | **shared** offset in the aggregated file |
| $FQ$ | **shared** queue of buffers to store data (free) |
| $WQ$ | **shared** queue of buffers to flush (write) |

copies of common variables used by all types of threads, or are shared variables. Variables shared across multiple threads are marked in their description.

#### 4.3.1. Senders

Sender threads transfer all node-local checkpoint data to the elected I/O leader in fixed-size chunks no larger than the pre-reserved buffers on the leader. When a buffer is filled, the sender initiates a blocking send call to the I/O leader. This process is repeated until all the local data has been sent to the leader. We could use non-blocking I/O between leaders and non-leaders (e.g., MPI_Isend), however it again does not provide any benefit due to size constraints on the I/O leaders. Thus, this results in a hurry-up-and-wait scenario, where senders still end up stalled. Instead, it is more beneficial to use blocking I/O, as it provides more opportunities to exert fine-grained control over local resource consumption like memory utilization or issuing I/O requests that prevent overwhelming leaders. We outline this workflow in Algorithm 2.

**Algorithm 2:** Sender Threads Overview

```
Input: S: copy of the aggregate size of local data being sent to an
       I/O leader
1 Function send_loop(S):
2     while S > 0 do
3         sz = min(BS, S)
4         B ← read in sz bytes of local data from offset l_o
5         blocking send ⟨B, sz⟩ to I/O leader
6         l_o+ = sz
7         S− = sz
```

#### 4.3.2. Receivers

Algorithm 3 outlines how receiver threads pipeline data from senders to writers. We use two queues to implement the circular buffer reuse protocol with pre-allocated buffers on the I/O leaders. When a buffer is in the *free* queue ($FQ$), it indicates its contents have been flushed and may be overwritten with incoming data from a follower. As such, when receivers are notified of a message, they check the *free* queue for a buffer. If one is available, they pop it off the front then complete the send; otherwise it waits until a buffer has been returned before accepting any incoming data. When a receive has been completed, the buffer gets added to the back of the *write* queue ($WQ$), where it waits to be flushed by the pool of writer threads. This process repeats until all the expected data from remote MPI ranks has been received.

#### 4.3.3. Writers

Writer threads flush data to the PFS in a loop until 2 conditions are met: (1) there is no more local data left to flush, (2) the *write* queue is empty *and* the receive thread has exited, indicating all expected data from followers has been flushed. To give the queue data priority (and prevent stalling remote MPI ranks), writer threads first check the *write* queue at the start of each loop, and only read in a chunk of local data if it is empty (line 4–7).

Similar to the *write* queue, we want writers to process local data in a FIFO order to mitigate load imbalance present in local checkpoint data, but without the need for a producer–consumer model. This is

**Algorithm 3:** Receiver Threads Overview

```
Input: R: aggregate size of data I/O leader will receive from all
       followers
1 Function recv_loop(R):
2     while R > 0 do
3         if FQ is empty then
4             wait
5         else
6             ⟨B, sz⟩ ← atomically pop front of FQ
7         ⟨B, sz⟩ ← sz bytes from follower
8         atomically add ⟨B, sz⟩ to back of WQ
9         R− = sz
```

because assigning local data to threads in uniform-sized chunks imposes strict boundaries that may not be optimal, as it is more difficult to take advantage of idle computational resources who may have finished faster (faster storage partition, luck, etc.). Algorithm 4 illustrates how we achieve this goal. Writers keep track of where the next local data chunk is via a shared data structure. The structure is atomically copied and updated when a thread needs to flush local data. Writers update and release control of the shared structure before copying over the local data to prevent it becoming a bottleneck.

Once a writer has an eligible buffer of data, it proceeds to flush the chunk of data to the PFS. Lines 10–12 of Algorithm 5 ensure each write to the aggregated file starts on a sequential, block-aligned address, in accordance with the optimized multi-threaded writes to the PFS discussion above. To facilitate the append-only writing approach, we use a similar idea to Algorithm 4 and keep track of the remote offset in an atomic variable (since there is only 1 destination file, each thread keeps a local copy of the file ID). Threads use the modulo function to correct any misalignment to the specified block size before writing it. As such, our implementation uses cheap synchronization points to allow threads to collaboratively process data. As such, we mitigate both global (from remote MPI ranks) and local (checkpoints from local MPI ranks) load imbalance.

**Algorithm 4:** Shared read

```
Output: ⟨B, sz⟩: buffer for local data and size descriptor
1 Function shared_read():
2     grab read lock
3     cp ← copy instance of shared, protected ⟨src, l_o⟩
4     sz ← min(BS, S)
5     ⟨src, l_o⟩ ← location of next chunk
6     S ← S − sz
7     release read lock
8     B ← sz bytes from local data at offset cp_o
9     return ⟨B, sz⟩
```

### 4.4. Implementation details

To implement our algorithms, we leverage the environment provided by VELOC [34], an asynchronous, multi-level checkpointing runtime. VELOC uses a client to expose a simple checkpointing API to the application. Applications invoke the checkpointing primitives when they are ready, which generates a request to the VELOC *client*. The client performs the local checkpoint, then alerts a post-processing engine running on each compute node (denoted the *active backend*) to asynchronously flush the checkpoint to the PFS in the background. Active backends use a highly modularized pipeline of targeted sub-protocols (e.g., error correction codes, checksum, threading, etc.) which can be configured at runtime to provide a lightweight and flexible flushing strategy.

As such, we extend VELOC with our own aggregation module, inserting it behind their optimized transfer module. The transfer module

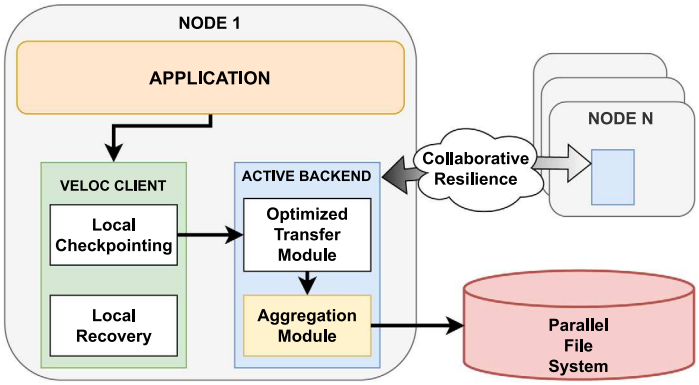


**Fig. 2.** A modified high level illustration [34] of how our aggregation module interacts with VELOC to persist distributed checkpoint data.

**Algorithm 5:** Write Overview

```
1  Function write_loop():
2      while True do
3          ⟨* b, sz⟩ ← ⟨null, 0⟩
4          if WQ not empty then
5              ⟨b, sz⟩ ← front of WQ
6          else
7              ⟨b, sz⟩ ← shared_read
8          if b is not null then
9              grab metadata lock
10             r_o ← normalize the offset (byte-align it)
11             cp_o ← make copy of shared remote offset r_o
12             r_o+ = sz
13             release metadata lock
14             write sz bytes from b to remote file at cp_o
15             if b came from WQ then
16                 atomically re-add b to FQ
17         while WQ is empty do
18             if recv thread exited then
19                 exit
20             wait
```

is usually responsible for performing the optimized flush to the PFS based on compile and runtime configurations (e.g., POSIX or direct I/O, threading capability, etc.). Thus, we have it invoke the aggregation module which then runs the algorithms presented above. We illustrate this in Fig. 2. As opposed to the native implementation, we modify the transfer module to use a client aggregator before calling the aggregation module. This blocks the backends from beginning the flush phase until all processes co-located on the same compute node have completed the local phase, which prevents straggling processes from trying to contribute checkpoint data after the leader election has taken place. Furthermore, we add support to specify a number of aggregation parameters (e.g., number of files ($K$), I/O threads, buffer sizes, etc.) at runtime via their configuration file, which is already used to enable and disable other modules supported by VELOC. For the purposes of this work, we disable all modules on the active backend except the optimized transfer module and our aggregation module via the configuration file.

## 5. Evaluation

*Experimental set-up.* Our experiments are performed on Oak Ridge National Lab's Frontier. Frontier is an ExaFLOP system with 9408 AMD compute nodes. Each compute node has 1 AMD EPYC 7453s 64 core CPU with 2 hardware threads per core. Compute nodes are connected in a dragonfly topology with a bisection bandwidth of 540 TB/s. Frontier is equipped with a Lustre parallel file system with a peak aggregated bandwidth of 14 TB/s (10 TB/s of NVMe, and 4.6 TB/s of HDD). The PFS is comprised of 1350 object storage targets (OSTs) managed by 450 I/O servers (OSS'). Frontier uses the linux-based Cray ecosystem. Our aggregation method is implemented on top of VELOC (v1.7). Our comparison strategies use GenericIO (Git tag 20190417), and ADIOS2 (v2.8.3). All code is compiled code using GCC 7.5.0 and cray-mpich 8.1.23.

*Software.* We compare our asynchronous I/O aggregation strategy implemented on top of VELOC to GenericIO [15] (GIO), an optimized implementation of MPI-IO; and against the high performance I/O library, ADIOS2 [12]. We discuss both in detail in Section 3. For the purposes of this work, we use GenericIO's non-collective version of MPI-IO, as our preliminary experiments show it provides higher performance at large-scale (> 1000 nodes). We use ADIOS2's BP5 transfer protocol, as this transfer engine provides many of the same configuration options as we do (number of I/O threads, size of memory buffers, number of files, block size, etc.), thus giving us a more appropriate comparison. We use the default *two level shm* aggregation strategy, which defines a number of MPI processes as aggregators that will write to disk, but will always have one per compute node [38].

### 5.1. Testing methodology

We evaluate the above aggregation strategies using both a custom checkpointing benchmark and with a real-world application. We use the synthetic benchmark to create a flexible checkpointing scenario that isolates I/O performance when flushing checkpoints to stable storage. Since asynchronous checkpoints are persisted concurrently with applications, we want to eliminate any resource contention which may impact results and skew initial analysis of our algorithms. We use the cosmological simulation, HACC [15], to evaluate our aggregation strategy in a real-world asynchronous checkpointing scenario (e.g., in the presence of a concurrent workload).

*Synthetic checkpointing benchmark.* The custom benchmark models checkpointing a distributed MPI application. It uses $C$ compute nodes and spawns $N = C * 8$ MPI processes, distributing 8 processes per compute node (since Frontier employs 8 GPUs per node, and HPC applications typically assign one process per GPU). To model load imbalance between checkpoints, we assign each process a 1 GiB ($\pm 20\%$) region of data. The benchmark invokes the necessary checkpointing primitives for the selected aggregation method and we time the results. For the asynchronous solutions (VELOC and ADIOS2), we isolate the local and flush checkpointing phases and limit our analysis only to the flush phase. Both VELOC and ADIOS2 provide blocking `flush` primitives that we utilize to accurately isolate and profile performance of only the flush phase to the PFS. In these experiments we set the

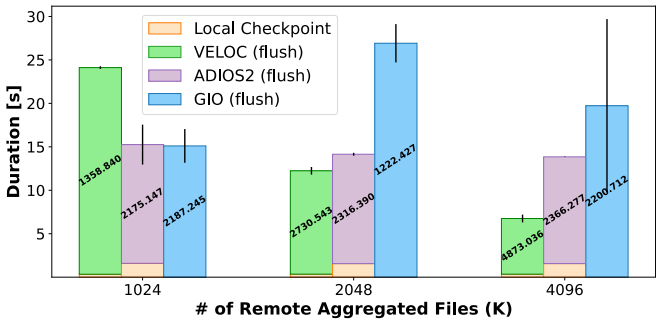


**Fig. 3.** Time it took to checkpoint the synthetic benchmark on 4096 compute nodes and writing to $K$ files. Lower is better.

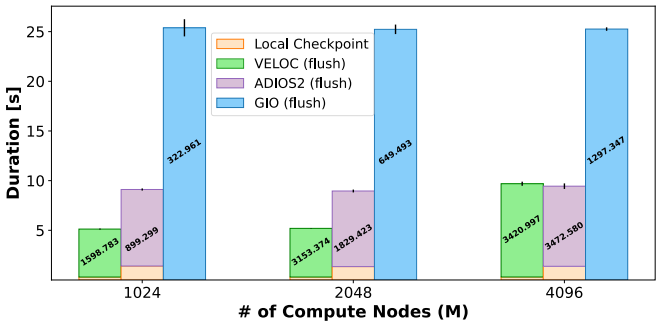


**Fig. 4.** Time it took to checkpoint the synthetic benchmark on $M$ compute nodes, where $K$ is at most 2048. The size of checkpoint data assigned to each MPI processes is uniform (e.g., tolerance = 0%). If $M < 2048$, $K = M$. Lower is better.

number of aggregated files, $K$, for each strategy (VELOC, ADIOS2, and GenericIO). Where we can (e.g., our aggregation strategy and ADIOS2), we set the buffer sizes to 64 MiB, and the number of I/O threads available to 8 (one for each local MPI process). GIO does not currently support limiting buffer sizes, and does not use multi-threading for I/O.

*HPC application: HACC.* For our real-world application, we use the high-performance cosmology code, HACC [15]. HACC is a complex parallel framework utilizing MPI that simulates mass evolution of the universe via particle-mesh techniques. HACC uses an in-situ analytics framework (CosmoTools), which provides a flexible checkpointing interface to the HACC application. HACC natively checkpoints with the GenericIO framework, and we use the plugin developed by the VELOC team [11] to checkpoint with our aggregated version of VELOC.

### 5.2. A weak scalability analysis of aggregated files

First, we characterize how the aggregation strategies perform as the ratio of $C : K$ grows. The point of this experiment is to capture how well the aggregation schemes scales as it gets more aggressive (e.g., more data per file). For these results, we keep the problem size fixed ($C = 4096$), and vary the number of aggregated files ($K$) between $\approx 1 - 4$ thousand. These results are shown in Fig. 3. The duration it takes to persist the entire checkpoint is displayed on the $y$-axis, which is further broken down by the local (orange) and flush (green, purple, and blue) phases. The throughput for the flush phase is calculated as described in Section 4 and printed in the middle of each bar. Since GIO is a synchronous write strategy, there is no local (orange) phase. Each experiment is ran 3 times and we average the results; the error bars illustrate the standard deviation across the 3 executions.

Overall, we see the performance for our aggregation solution drops by about half as the aggregation ratio grows (right to left). Our previous works [37] found that compute nodes on frontier are bound by the network interface card (NIC), achieving an outgoing write throughput of no more than $\approx 2.5$ GiB/s per node, regardless of the number of I/O threads. Therefore, the theoretical peak of our strategy is $K * 2.5$ GiB/s. The left-most cluster of bars shows the most extreme aggregation ratio of $4 : 1$. Both ADIOS2 and GenericIO's non-collective write-at-offset as implemented by MPI-I/O achieves $\approx 1.5\times$ higher throughput than ours. Thus, these results illustrate the fine trade-off between managing the number of concurrent writers, and under-utilizing the available I/O resources. In our case, we lose some performance by first funneling data through the proxies even though the file system has available resources to spare at this scale.

However, as the number of files (left to right) increases, our aggregation solution exhibits the most improved performance. When the aggregation ratio is $2 : 1$ ($K = 2048$), our strategy performs $\approx 2.20\times$ better than GenericIO, and $\approx 1.16\times$ greater throughput than ADIOS2. This continues to scale as the aggregation ratio converges to 1 (e.g., $K = 4096$ and each compute node is an I/O leader), where we get more than $2\times$ better performance than both GenericIO and ADIOS2. Furthermore, this scenario results in the most fair comparison between our strategy and ADIOS2, since they employ a writer on all compute nodes and this test case results in the same configuration for our modified version of VELOC. When the aggregation ratio is 1, we argue that our method performs better due to our simple, pipelined file transfer protocol. The BP5 format used by ADIOS2 suffers similar bottlenecks to HDF5 discussed in Section 3 from self-descriptive metadata management and data layout, thereby resulting in lower performance. Furthermore, we note that our strategy maintains a low standard deviation across all scenarios, which demonstrates its stability.

### 5.3. Characterizing the impact of load imbalance on asynchronous aggregation

In these next experiments, our goal is to characterize the effect of load imbalance. We quantify the impact of our globally-balanced partitioning strategy described in Section 4.2, and characterize the trade-off incurred by implementing thread synchronization techniques to achieve thread-level load-balancing on the I/O leader. In these experiments, we vary the checkpoint size assigned to each process by $\pm$ 0,10, and 20%. We keep $K = 2048$, and vary $C = \{1024, 2048, 4096\}$. We present the findings of these results in Figs. 4- 6.

Fig. 4 uses uniform checkpoint sizes across all application processes (e.g., size varies by $\pm 0\%$). Consistent with our experiments in Fig. 3, when the size of all checkpoints is exactly equal to 1 GiB and the ratio of $C : K$ is 1 (left-most and middle clusters), we obtain $\approx 2\times$ the performance of ADIOS2, and $\approx 5\times$ greater throughput than GIO. When the ratio of $C : K$ is 2 (right-most cluster), our throughput drops by $\approx 2\times$, again because our interactions to the PFS are now funneled through half the number of compute nodes. In the same scenario, ADIOS2, obtains $\approx 1.08\times$ higher throughput than we do, likely because evenly sized checkpoints of exactly 1 GiB align to the allocation unit of the file system; compounded with higher utilization of PFS resources, they obtain better performance.

Figs. 5 and 6 show the results when we vary the size of checkpoints $\pm 10$ and $\pm 20\%$, respectively. When $C \leq 2048$, our strategy maintains similar performance compared to uniform scenario presented in Fig. 4. This shows that our global partitioning strategy suffers negligible degradation from both global and local load imbalance. The marginal increase ($< 10\%$) in flush duration is due to the fact that we have to fix the offset in the remote file more times when the checkpoints are not uniform (whereas in the case of 0% tolerance writes are always block-aligned, since our buffer size is a factor of the allocation unit on Frontier). When we set $C = 4096$, it takes no more than $2.2\times$ longer to complete the checkpoint. The ideal value would be only $2\times$ the duration it took in the previous scenario, however, because in this scenario we are also processing received data, we have to correct offsets for the remote file roughly $2\times$ more. This is supported

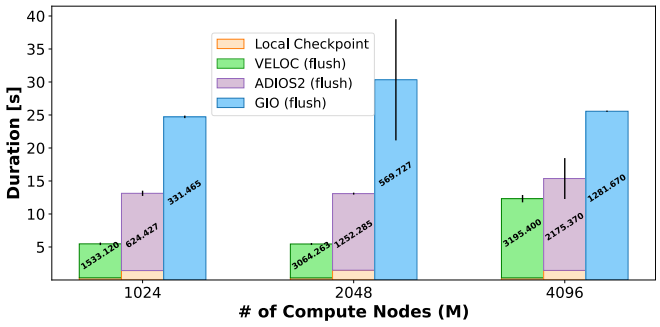


**Fig. 5.** Time it took to checkpoint the synthetic benchmark on $M$ compute nodes, where $K$ is at most 2048. The size of checkpoint data assigned to each MPI processes varies by 10%. If $M < 2048$, $K = M$. Lower is better.

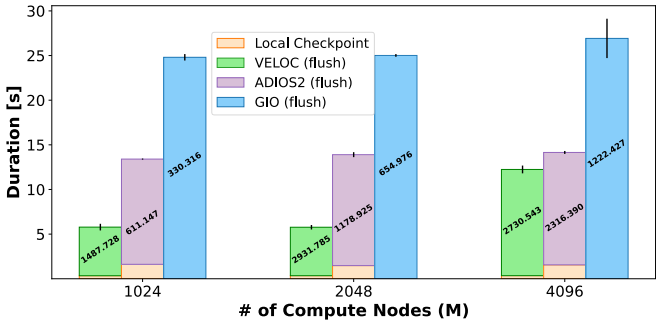


**Fig. 6.** Time it took to checkpoint the synthetic benchmark on $M$ compute nodes, where $K$ is at most 2048. The size of checkpoint data assigned to each MPI processes varies by 20%. If $M < 2048$, $K = M$. Lower is better.

by our aggregation strategy maintaining the same amount of overhead, regardless of the variation between checkpoints (10 or 20%). GIO also maintains pretty similar performance across the different scenarios despite varying checkpoint sizes, however, they are still $\approx 2\times$ slower than our I/O aggregation algorithm. ADIOS2, on the other hand, is not as robust against load imbalance, taking $\approx 1.65\times$ longer to flush checkpoints when the size between checkpoints varies by 20% as opposed to the uniformly sized checkpoints. These results highlight how aggregation strategies suffer lower performance when load balance is untreated. Similar to Fig. 3, our aggregation strategy continues to maintain a low standard deviation compared to the other two, even in the presence of load imbalance.

### 5.4. Comparing I/O leaders: A fine-grained breakdown

We isolate performance metrics of our aggregation strategy across various I/O leaders. In Fig. 7, we zoom in on the fastest and slowest I/O leaders, along with 5 other randomly sampled leaders. Here, we are interested in characterizing the maximum and average imbalance across I/O leaders. For these results, we look at the most extreme case when the size of data per checkpoint varies by $\pm 20\%$. The results presented in these figures use $C = 4096$ compute nodes, and are aggregating to $K = 2048$ files. Furthermore, we isolate the performance of flushing received (queue), and local data to identify if and which is a consistent bottleneck across I/O leaders.

The dotted line in Fig. 7 shows the average checkpoint time across all the leaders, which illustrates how far each of the sampled leaders is from the average. We isolate the time I/O leaders spent writing local (blue) and received (orange) data and print the corresponding throughput for each source in the center of the bar. Each leader is identified by their MPI rank in the VELOC backend communicator. In Fig. 7, the aggregated throughput for each I/O leader (with the exception being the slow leader) is $\approx$ 2 GB/s, which we previously found to be the maximum outgoing bandwidth of a compute node on Frontier, as discussed in Section 5.2. Thus, this shows that our I/O leaders are typically able to maximize their outgoing throughput, even in the presence of both local (checkpoint data per local file) and group-wide (checkpoint data per compute node) load imbalance. I/O leaders achieve sufficiently similar performance, with the slow I/O leader no more than 12% slower than the average. We investigate this more in-depth in the next two figures.

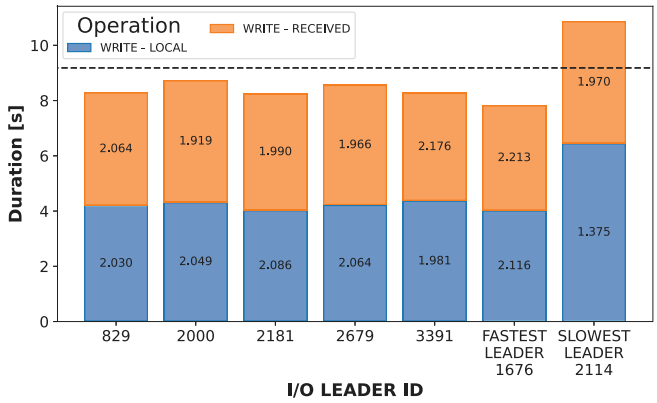


**Fig. 7.** Breakdown of time spent writing received (orange) versus local (blue) data on the fastest, slowest, and 5 other randomly sampled leaders. Shorter bars are better. The values in the middle of each bar is the throughput, higher values are better.

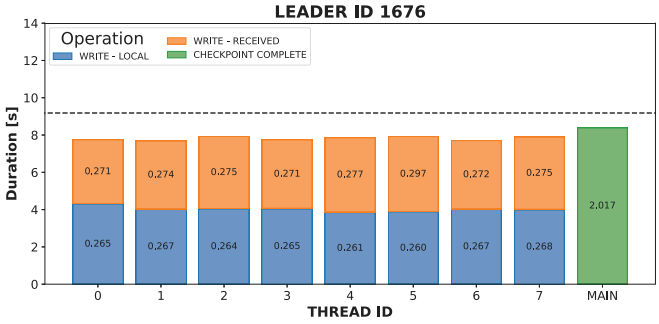


**Fig. 8.** Breakdown of the time it took each thread on the fastest leader to flush data. The green bar is the maximum time it took for *all* threads on the leader to exit (e.g., bound by the slowest thread). Lower is better.

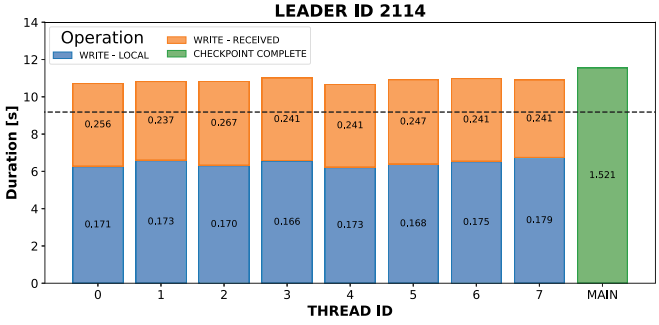


**Fig. 9.** Breakdown of the time it took each thread on the slowest leader to flush data.

Figs. 8 and 9 present thread-level breakdowns comparing the fast and slow leader's flush phase. We only present the aggregate amount of time each thread spent on POSIX `write` calls. There are other operations performed during the checkpoint phase (e.g., waiting for access to the receiver queue, atomically accessing the shared data structures, reading in local data, etc.), however, we find these to be

negligible ($\approx 10^{-5}$ s), and are thus omitted for clarity. To support this claim, the green bar in both graphs represents the duration the main thread spent in the aggregation module (as it can only return until all threads have finished). Thus, we show that threads spent the bulk of their time performing I/O with the PFS. One purpose of these breakdowns is to illustrate how effectively our threads are at managing load imbalance. Even though distributed checkpoints (including those on the same compute node) exhibit a ±20% variability in size, since threads process data in a collective fashion, we observe none of them becoming a bottleneck on either leader. This illustrates how our aggregation design implicitly maximizes the work assigned to threads even in the presence of load imbalance, to maximize performance (≈2 GB/s per compute node).

Secondly, we are interested in understanding why slow I/O leaders take almost 2× longer to flush local data compared to the received data. In Fig. 8, writer threads had similar performance when flushing local and received data, which is important to highlight as it shows there is no explicit bottleneck in the algorithms that process local data. On the slow leader, we see that local data incurs a 43% lower overall throughput across *all* the writer threads. At a glance, it may seem that is due to load imbalance between the data assigned to the I/O leader and the non-leader compute node in its group. However, we observed this difference to be ≈ 600 MB on the slow leader, compared to ≈ 200 MB on the fast leader. Thus, it is an unlikely explanation. Instead, it is more probable that the observed behavior on the slow leader is due to how we prioritize received data over the local data. Once non-leaders are up and running they are consistently supplying the I/O leader with data to flush, which effectively serializes flushing the local data behind all of the received data. This may be enough to fill up OS-level cache/buffers, thereby resulting in lower throughput for the data that is flushed secondary (local data). Another explanation is that the metadata servers or storage targets assigned to the slow leader experience a period of higher competition from another workload on the system, thereby limiting its availability and lowering the throughput. Something like this is more likely to effect either mostly the received data or mostly the local data depending on when it happens. Overall, testing in a more controlled environment (e.g., full machine reservation), is needed to isolate the root cause.

### 5.5. Aggregated checkpointing impact on concurrent workload (HACC)

We evaluate our file aggregation strategy in a real-world checkpointing scenario using the scientific application, HACC. We use 2048 compute nodes for a total of 16,384 MPI processes to simulate 14,464 particles. In these experiments, the aggregation implementations (our VELOC implementation, and GenericIO) set the number of remote files $K = 1024$, the same ratio of files to compute nodes as our microbenchmark experiments presented in above. HACC generates roughly the same amount of data per MPI process with some variance, similar to our tests that vary checkpoint sizes by ≈ 20%. We also capture the runtime of the HACC application when checkpointing is not utilized, thereby measuring how checkpointing strategies impact the application. We compare our aggregation strategy against GenericIO and VELOC's one-file-per-process flushing strategy to provide a holistic overview of how our aggregation strategy compares to state-of-the-art aggregated checkpointing, and how it impacts the efficacy of VELOC.

We directly compare performance and impact on the application between the 4 checkpointing schemes in Fig. 10. The black numbers printed at the top of each bar is the total runtime of the simulation captured via the `time` system call when launching the application. Since GenericIO is synchronous, we subtract the time spent flushing the checkpoint (blue) from the runtime, which gives us time spent on computation (green), and stack the two bars, as they run in serial. Since VELOC's local phase is serialized with the application, we subtract the time spent in this phase (reported by HACC) from the runtime. VELOC flushes checkpoints concurrently with the application, so the

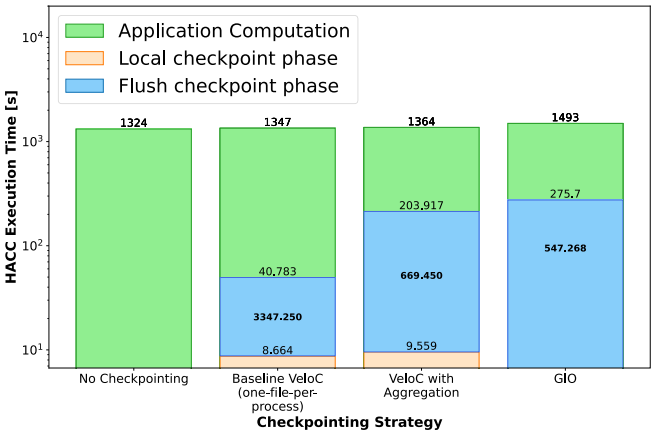


**Fig. 10.** Operational breakdown of the HACC simulation under different checkpointing strategies on 2048 compute nodes, and aggregating to 1024 files (using our modified version of VELOC and GIO). Lower is better.

flush phases (blue) are overlaid onto the computation (green). The bold numbers printed in the center of the blue bars quantify the aggregated throughput when flushing the checkpoint to the PFS.

Comparing our aggregated implementation of VELOC against the baseline one-file-per-process, the aggregation strategy takes ≈ 4× longer in the flush phase, and gets ≈ 5× lower throughput than the one-file-per-process implementations. Based on our microbenchmark experiments presented in Fig. 3, 1024 compute nodes are not enough to fully saturate the PFS resources attached to Frontier, meanwhile, VELOC's file-per-process strategy does. Thus, they are able to perform more parallel and lock-free I/O, which ultimately provides exceptionally high throughput (close to 70% of the HDD peak aggregated bandwidth attached to Frontier). While file-per-process checkpointing gets higher performance, we still meet the goals of our work, as outlined in Section 2. Specifically, we are able to reduce the number of checkpoint files by 16×, while introducing a < 2% overhead to the total runtime of the application (13 s).

Meanwhile, checkpointing with GenericIO introduces a ≈ 12% checkpointing overhead to the application, as opposed to our 3%. Furthermore, we are able to flush data to the PFS 1.2× faster than GenericIO, even while sharing resources with the application. Overall, our results show that when adopting our aggregation strategy, VELOC can achieve the same level of file compactness as leading state-of-the-art synchronous checkpointing strategies with minimal performance overhead introduced to the concurrently running application compared to VELOC's baseline performance.

### 5.6. Concurrent workload (HACC) impact on asynchronous aggregation

Figs. 11–13, illustrate the same performance breakdowns done for the microbenchmark in Figs. 7–9, and help characterize how our aggregation strategy is impacted by a concurrently running workload. Compared to our microbenchmark experiments in Fig. 7, Fig. 11 shows that a concurrent workload like HACC degrades the flushing throughput of our aggregation strategy by almost 2× in a similar scenario (e.g., 1:1 ratio of leaders to non-leaders). Thus, these experiments illustrate how susceptible this aggregation strategy is to scarcity in the network bandwidth. This creates an important trade-off for application developers to consider, as prioritizing application work results in almost negligible runtime overhead (< 3%) at the expense of a longer phase. However, if aggregation causes the flush phase to extend too long where applications cannot efficiently capture subsequent checkpoints, it is worth it to increase the priority checkpointing work (since the application will be blocked anyway).

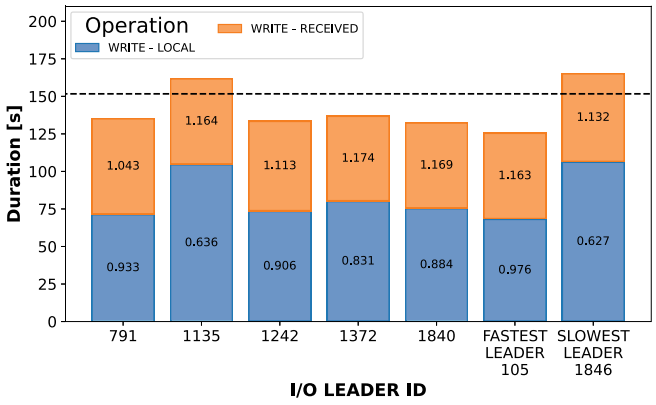


**Fig. 11.** Breakdown of time spent writing received (orange) versus local (blue) data on the fastest, slowest, and 5 other randomly sampled leaders when checkpointing the HACC application. Shorter bars are better. The values printed in the middle of each bar is the throughput, higher values are better.

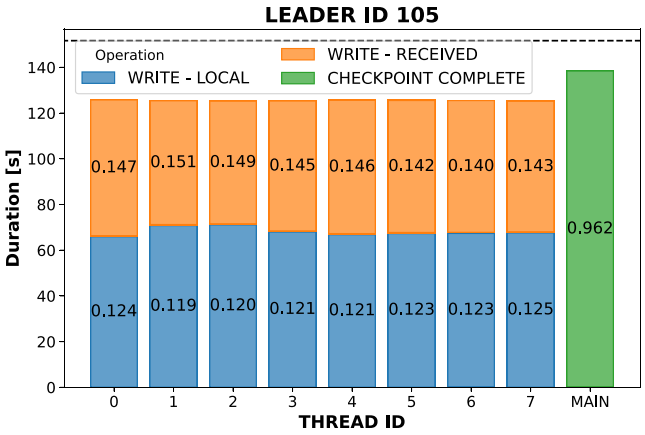


**Fig. 12.** Breakdown of the time it took each thread on the fastest leader to flush data. The green bar is the maximum time it took for *all* threads on the leader to exit (e.g., bound by the slowest thread). Shorter bars are better.

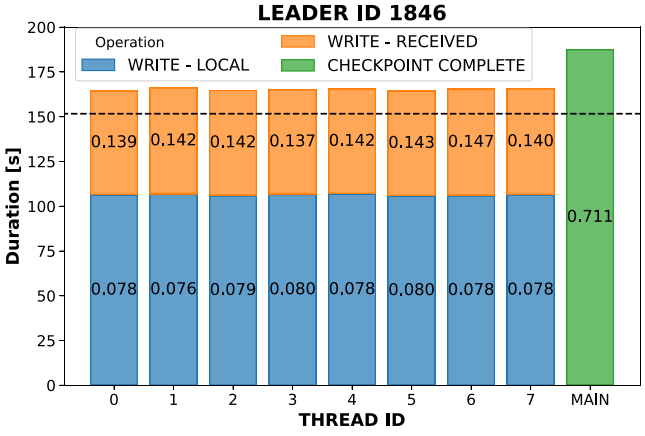


**Fig. 13.** Breakdown of duration it took each thread on the slowest leader to flush received (orange) and local (blue). Lower is better.

In Fig. 7, we observed that slow I/O leaders took 2× longer to flush local data compared to remote (queue) data. We see this same trend in Fig. 11, however, we also start to observe a more mild version of this degradation across *all* the I/O leaders. Given that we start to see this trend appear across all sampled I/O leaders, it supports our hypothesis that such variable performance when writing either local or received data is most likely attributed to outside factors impacting the file system at a specific time. The reason this trend becomes more visible in these results is because the checkpoint duration in the HACC experiments take ≈ 40× longer compared to the microbenchmark experiments (due to data volume). Thus, performance degradation is likely due to PFS resource availability at that time, rather than caused by a bottleneck when processing local data (e.g., synchronizing around shared data structures).

Compared to the microbenchmark experiments in Fig. 11, the difference between the average and slowest I/O leader is roughly the same (≈ 12% in Fig. 7 and ≈ 16% in Fig. 11). This confirms a key conclusion from Section 5.4: that global load balancing strategies (e.g., evenly sized groups of compute nodes) are still effective even in the presence of local load imbalance (differently sized local checkpoint files). Thus, it may not be worth it to guarantee uniform load balance, especially given that I/O leaders themselves can experience variable performance as discussed in the previous paragraph. Instead, it may be more beneficial for future works to explore *adaptable* aggregation techniques, where data from slow leaders is rerouted to fast leaders to better utilize I/O leaders, and mitigate performance degradation.

Figs. 12 and 13 present thread-level breakdowns of the flush phase on the fast and slow I/O leaders, respectively. Compared to Figs. 8 and 9, threads in these results (Figs. 12 and 13) show a bit more uniformity on both leaders, highlighting how dependent performance of background I/O is on local resource availability (e.g., network bandwidth, CPU cores, memory space, etc.). HACC frequently needs to exchange data across compute nodes, and utilizes GPUs which requires frequent data transfers. Thus, our I/O operations will suffer latency to share bandwidth, especially as a background operation.

Furthermore, we see a similar trend that we noted in Figs. 8 and 9, where I/O threads on the slow leader are getting marginally lower performance when flushing received data (averaging closer to the lower end of 140 MB/s), compared to the fast leader (averaging closer to 150 MB/s). This continues to support our analysis that slow leaders are a symptom of the PFS resources they get allocated, rather than a design bottleneck. Furthermore, the results in Figs. 12 and 13 show how all I/O threads across both leaders begin to experience degraded performance when flushing the local data, illustrating that I/O performance to the PFS is degrading as a whole. As in the microbenchmark experiments, slow I/O leaders are not consistent across multiple test runs, indicating there is no cornerstone in our design that generates this behavior.

## 6. Conclusions

We design and develop a high-performance, scalable I/O aggregation strategy for asynchronous multi-level checkpointing. Unlike state-of-art I/O aggregation approaches that feature a decoupled two-phase I/O strategy (involving a separate collection of data on proxies that subsequently write the data to a parallel file system), we introduce several innovative design principles targeting asynchronous I/O performance using background threads that do not block the application. Specifically, we discuss the importance of autonomous group-based leader election, streamlined producer–consumer flushing, optimized multi-threaded writes to parallel file systems and load balancing I/O strategies to avoid stragglers.

We run extensive experiments at scale using both micro-benchmarks and a real-life application (HACC, a large-scale cosmology simulation). Results show our approach achieves ≈ 2× higher throughput than aggregated synchronous checkpointing (notably GenericIO). Furthermore, compared with I/O libraries that support asynchronous aggregation (notably ADIOS2), our approach achieves 1.16 × –2× higher I/O flush

throughput in the background as the number of checkpoint files is close to the number of nodes. On the other hand, when the number of checkpoint files is significantly less than the number of compute nodes (i.e., 1:4), ADIOS2 scales better as it allows more than one process to write to the same shared file (as opposed to our approach, which allows only one leader per checkpoint file). Under these circumstances, our approach achieves a good trade-off between reducing the number of checkpoint files compared to one-file-per-process solutions, while maintaining a high I/O flush throughput at scale, especially on modern HPC Exascale systems (notably Frontier).

Encouraged by these results, we plan to improve our I/O aggregation strategies in several directions. First, our experiments have revealed that a single leader writing to a large checkpoint file that aggregates the checkpointing data of many MPI ranks makes suboptimal use of the I/O bandwidth to the PFS at scale, even when the number of leaders is larger than the number of I/O servers. Based on these observations, it seems more than one leader is needed to saturate each I/O server. Thus, we plan to investigate what is the optimal number of leaders needed to extract the highest overall I/O flush throughput, and how does this vary (if at all) at scale. Then, based on this investigation, we will design and develop adaptive leader election strategies and their mapping to I/O servers. Second, we did not investigate in detail the competition for resources between the HPC applications and background I/O threads performing the aggregation and flushing of checkpointing data. While a large number of modern HPC applications make use of GPUs for efficient computations (and therefore can spare generous amounts of CPU and host memory), other resources like network bandwidth used by the application for communications may compete with the communication performed by the I/O background. We will investigate the interference caused by competing network communication and will design mitigation strategies accordingly.

## CRediT authorship contribution statement

**Mikaila J. Gossman:** Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Bogdan Nicolae:** Writing – review & editing, Supervision, Resources, Funding acquisition, Formal analysis, Conceptualization. **Jon C. Calhoun:** Writing – review & editing, Supervision, Funding acquisition, Conceptualization.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Jon C. Calhoun reports financial support was provided by National Science Foundation. Bogdan Nicolae reports financial support was provided by US Department of Energy. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Data will be made available on request.

## Acknowledgments

Clemson University, Argonne National Lab, and Oak Ridge National Lab are acknowledged for generous allotment of compute time on the Palmetto cluster, Theta supercomputer, and Frontier supercomputer, respectively. This work was supported by the National Science Foundation, USA [SHF-1910197 and SHF-1943114]; the U.S. Department of Energy, Office of Science [DE-AC02-06CH11.357].

## References

[1] H. Zhang, E.M. Constantinescu, Optimal checkpointing for adjoint multistage time-stepping schemes, J. Comput. Sci. 66 (2023) 101913.
[2] T.L. Scao, A. Fan, C. Akiki, et al., BLOOM: A 176B-parameter open-access multilingual language model, 2022, http://dx.doi.org/10.48550/arXiv.2211.05100, arXiv e-prints. arXiv:2211.05100.
[3] B. Neyshabur, H. Sedghi, C. Zhang, What is being transferred in transfer learning? in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems, Vol. 33, Curran Associates, Inc, 2020, pp. 512–523, URL https://proceedings.neurips.cc/paper_files/paper/2020/file/0607f4c705595b911a4f3e7a127b44e0-Paper.pdf.
[4] A.P. Thompson, H.M. Aktulga, R. Berger, D.S. Bolintineanu, W.M. Brown, P.S. Crozier, P.J. in 't Veld, A. Kohlmeyer, S.G. Moore, T.D. Nguyen, R. Shan, M.J. Stevens, J. Tranchida, C. Trott, S.J. Plimpton, LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales, Comp. Phys. Comm. 271 (2022) 108171, http://dx.doi.org/10.1016/j.cpc.2021.108171.
[5] E. Rojas, A.N. Kahira, E. Meneses, L.B. Gomez, R.M. Badia, A study of checkpointing in large scale training of deep neural networks, 2020, arXiv preprint arXiv:2012.00825.
[6] B. Nicolae, J. Li, J.M. Wozniak, G. Bosilca, M. Dorier, F. Cappello, Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models, in: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID, IEEE, 2020, pp. 172–181.
[7] Lustre : A scalable , high-performance file system cluster, 2003.
[8] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, M. Snir, Toward exascale resilience: 2014 update, Supercomput. Front. Innov. 1 (1) (2014) 5–28, http://dx.doi.org/10.14529/jsfi140101, URL https://superfri.org/index.php/superfri/article/view/14.
[9] D. Dauwe, S. Pasricha, A.A. Maciejewski, H.J. Siegel, An analysis of multilevel checkpoint performance models, in: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, 2018, pp. 783–792, http://dx.doi.org/10.1109/IPDPSW.2018.00125.
[10] M. Gholami, F. Schintke, Combining XOR and partner checkpointing for resilient multilevel checkpoint/restart, in: 2021 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2021, pp. 277–288, http://dx.doi.org/10.1109/IPDPS49936.2021.00036.
[11] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, F. Cappello, VeloC: Towards high performance adaptive asynchronous checkpointing at large scale, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2019, pp. 911–920, http://dx.doi.org/10.1109/IPDPS.2019.00099.
[12] W.F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, S. Klasky, ADIOS 2: The adaptable input output system. A framework for high-performance data management, SoftwareX 12 (2020) http://dx.doi.org/10.1016/j.softx.2020.100561.
[13] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B.R. de Supinski, S. Matsuoka, Design and modeling of a non-blocking checkpointing system, in: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–10, http://dx.doi.org/10.1109/SC.2012.46.
[14] K. Mohror, A. Moody, G. Bronevetsky, B.R. de Supinski, Detailed modeling and evaluation of a scalable multilevel checkpointing system, IEEE Trans. Parallel Distrib. Syst. 25 (09) (2014) 2255–2263, http://dx.doi.org/10.1109/TPDS.2013.100.
[15] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, et al., HACC: Simulating sky surveys on state-of-the-art supercomputing architectures, New Astron. 42 (2016) 49–65, http://dx.doi.org/10.1016/j.newast.2015.06.003.
[16] T.Z. Islam, K. Mohror, S. Bagchi, A. Moody, B.R. de Supinski, R. Eigenmann, MCREngine: A scalable checkpointing system using data-aware aggregation and compression, in: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–11, http://dx.doi.org/10.1109/SC.2012.77.
[17] S.-M. Tseng, B. Nicolae, F. Cappello, A. Chandramowlishwaran, Demystifying asynchronous I/O interference in HPC applications, Int. J. High Perform. Comput. Appl. 35 (4) (2021) 391–412, http://dx.doi.org/10.1177/10943420211016511.
[18] A. Maurya, R. Underwood, M.M. Rafique, F. Cappello, B. Nicolae, DataStates-LLM: Lazy asynchronous checkpointing for large language models, in: The 33rd International Sym- Posium on High-Performance Parallel and Distributed Computing, HPDC'24, 2024, http://dx.doi.org/10.1145/3625549.36586857.
[19] M. Sato, Y. Kodama, M. Tsuji, T. Odajima, Co-design and system for the supercomputer "Fugaku", IEEE Micro 42 (02) (2022) 26–34, http://dx.doi.org/10.1109/MM.2021.3136882.
[20] H. Khetawat, C. Zimmer, F. Mueller, S. Atchley, S.S. Vazhkudai, M. Mubarak, Evaluating burst buffer placement in HPC systems, in: 2019 IEEE International Conference on Cluster Computing, CLUSTER, 2019, pp. 1–11, http://dx.doi.org/10.1109/CLUSTER.2019.8891051.

[21] J. Rosario, R. Bordawekar, A. Choudhary, Improved parallel I/O via a two-phase run-time access strategy, ACM SIGARCH Comput. Archit. News 21 (1993) 31–38, http://dx.doi.org/10.1145/165660.165667.
[22] F. Tessier, V. Vishwanath, E. Jeannot, TAPIOCA: An I/O library for optimized topology-aware data aggregation on large-scale supercomputers, in: 2017 IEEE International Conference on Cluster Computing, CLUSTER, 2017, pp. 70–80, http://dx.doi.org/10.1109/CLUSTER.2017.80.
[23] R. Thakur, W. Gropp, E. Lusk, Data sieving and collective I/O in ROMIO, in: Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation, 1999, pp. 182–189, http://dx.doi.org/10.1109/FMPC.1999.750599.
[24] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, M. Winslett, Server-directed collective I/O in panda, in: Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, 1995, p. 57, http://dx.doi.org/10.1109/SUPERC.1995.241778.
[25] A. Ramos Carneiro, J.L. Bez, F. Zanon Boito, B. Alves Fagundes, C. Osthoff, P.O. Navaux, Collective I/O performance on the santos dumont supercomputer, in: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP, 2018, pp. 45–52, http://dx.doi.org/10.1109/PDP2018.2018.00015.
[26] R. Thakur, W. Gropp, E. Lusk, Data sieving and collective I/O in ROMIO, in: Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation, 1999, pp. 182–189, http://dx.doi.org/10.1109/FMPC.1999.750599.
[27] Y. Tsujita, K. Yoshinaga, A. Hori, M. Sato, M. Namiki, Y. Ishikawa, Multithreaded two-phase I/O: Improving collective MPI-IO performance on a lustre file system, in: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2014, pp. 232–235, http://dx.doi.org/10.1109/PDP.2014.46.
[28] R. Feki, E. Gabriel, Design and evaluation of multi-threaded optimizations for individual MPI I/O operations, in: 2022 30th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP, 2022, pp. 122–126, http://dx.doi.org/10.1109/PDP55904.2022.00027.
[29] Q. Kang, S. Lee, K. Hou, R. Ross, A. Agrawal, A. Choudhary, W.-k. Liao, Improving MPI collective I/O for high volume non-contiguous requests with intra-node aggregation, IEEE Trans. Parallel Distrib. Syst. 31 (11) (2020) 2682–2695, http://dx.doi.org/10.1109/TPDS.2020.3000458.
[30] Q. Jensen, F. Jagodzinski, T. Islam, FILCIO: Application agnostic I/O aggregation to scale scientific workflows, in: 2021 IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC, 2021, pp. 1587–1592, http://dx.doi.org/10.1109/COMPSAC51774.2021.00236.
[31] M. Folk, G. Heber, Q. Koziol, E. Pourmal, D. Robinson, An overview of the HDF5 technology suite and its applications, in: AD'11: The 2011 Workshop on Array Databases, Association for Computing Machinery, Uppsala, Sweden, 2011, pp. 36–47.
[32] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, S. Matsuoka, FTI: High performance fault tolerance interface for hybrid systems, in: SC '11: The 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, USA, 2011, pp. 32:1–32:32.
[33] I.S. Reed, G. Solomon, Polynomial codes over certain finite fields, J. Soc. Ind. Appl. Math. 8 (2) (1960) 300–304.
[34] B. Nicolae, A. Moody, G. Kosinovsky, K. Mohror, F. Cappello, VELOC: VEry low overhead checkpointing in the age of exascale, 2021, CoRR. arXiv:2103.02131.
[35] S. Atchley, Checkpointing tips, 2023, https://www.olcf.ornl.gov/wp-content/uploads/Checkpointing-Tips-OLCF-User-Call-20230329.pdf. (Accessed 31 August 2023).
[36] T. Herault, Y. Robert, A. Bouteiller, D. Arnold, K. Ferreira, G. Bosilca, J. Dongarra, Checkpointing strategies for shared high-performance computing platforms, Int. J. Netw. Comput. 9 (1) (2019) 28–52, URL http://ijnc.org/index.php/ijnc/article/view/195.
[37] M. Gossman, B. Nicolae, J. Calhoun, Modeling multi-threaded aggregated I/O for asynchronous checkpointing on HPC systems, in: ISPDC'23: The 22nd IEEE International Conference on Parallel and Distributed Computing, Bucharest, Romania, 2023, pp. 101–105.
[38] Aggregation. https://adios2.readthedocs.io/en/v2.9.2/advanced/aggregation.html.

**Mikaila** received her bachelors degree in Computer Engineering in May 2020 from Clemson University's Holcombe Department of Electrical and Computer Engineering, where she is now a third year Ph.D. student. Her research interests are centered around improving the performance and scalability of scientific applications in High Performance Computing (HPC) systems, with a focus on checkpoint-restart (C/R) and I/O scalability. Currently, she is working in collaboration with Argonne National Lab as part of the VELOC team.

**Bogdan Nicolae** is a Computer Scientist with Argonne National Laboratory (Chicago, USA) and Research Professor at Illinois Institute of Technology (Chicago, USA). He specializes in scalable storage, data management and fault tolerance for large scale distributed systems, in particular at the intersection of high performance computing, big data analytics and artificial intelligence. He is interested by and authored numerous papers in areas such as checkpoint-restart, state capture and migration, data and metadata decentralization and high availability, concurrency control in data management, multi-versioning and historic access, declarative data models, live migration. He is a regular PC member and participates in the organization of major international conferences around parallel and distributed systems: SC, IPDPS, HPDC, CCGrid, CLUSTER, ICS, HIPC, ICDCS, ICPP, EuroPar, EuroMPI, etc. He is a regular reviewer for journals such as: TPDS, JPDC, FGCS, PARCO, TC, TCC, IJHPCA.

**Jon** is an Assistant Professor of Electrical and Computer Engineering and the director of the Future Technologies in Heterogeneous and Parallel Computing (FTHPC) Laboratory at Clemson University. I obtained a Ph.D. in Computer Science from University of Illinois at Urbana-Champaign under the direction of Professors Luke Olson and Marc Snir. At Clemson, my research interests broadly lie in two areas: fault tolerance and data compression. In particular, I am interested in fault tolerance issues related to high-performance computing systems such as improving checkpoint-restart and understanding the impact of silent data corruption HPC applications. With respect to data compression, I focus on developing and applying novel lossy and lossless data compression algorithms in many different areas from high-performance computing to intelligent transportation systems to mitigate bandwidth and storage bottlenecks